Exp Astron (2013) 35:131–155
DOI 10.1007/s10686-011-9276-8

ORIGINAL ARTICLE

Tracing and using data lineage for pipeline processing in Astro-WISE

Johnson Mwebaze · Danny Boxhoorn · Edwin A. Valentijn

Received: 16 June 2011 / Accepted: 23 November 2011 / Published online: 13 December 2011

Abstract Most workflow systems that support data provenance primarily focus on tracing lineage of data. Data provenance by data lineage provides the derivation history of data including information about services and input data that contributed to the creation of a data product. We show that tracing lineage by means of full backward chaining not only enables users to share, discover and reuse the data, but also supports scientific data processing through storage, retrieval and (re)processing of digitized scientific data. In this paper, we present Astro-WISE, a distributed system for processing, analyzing and disseminating wide field imaging astronomical data. We show how Astro-WISE traces lineage of data and how it facilitates data processing, retrieval, storage and archiving. Particularly we show how it solves issues related to the changing data items typical for the scientific environment, such as physical changes in calibrations, our insight in these changes and improved methods for deriving results.

Keywords Data lineage · Provenance · Pipeline data processing · Astro-WISE

1 Introduction

The nature of today's scientific experiments requires innovative dynamic approaches, in which results can be disseminated, re-derived, customized to each user's specific needs and shared between research groups. A scientific

J. Mwebaze (✉) · D. Boxhoorn · E. A. Valentijn
Kapteyn Astronomical Institute, University of Groningen,
Landleven 12, 9700 AV Groningen, The Netherlands
e-mail: jmwebaze@cit.ac.ug

data analysis work cycle consists of basic actions: create (or select) an analysis pipeline and execute its initial run; modify input data or an analysis function and propagate the change; add an analysis function and re-execute the pipeline; and create related pipelines based on an abstract workflow while publishing results to a centralized data store.

Given the complexity of scientific pipelines coupled with distributed data and resources, the process of building the required analysis pipeline is not trivial. Like-wise without data lineage (or provenance): it is difficult to know which pipeline produced valid results; it is difficult to verify the correctness of data; and also difficult to follow through the derivation process to find out the cause of an abnormality that could have been discovered in the data.

The problem of tracing lineage of data has been extensively studied in databases [4, 6] and in workflows [1, 7, 8, 12, 13] in the context of providing derivation history of data. Provenance aware systems support mechanisms by which execution can be recorded from which the provenance of results can later be determined. Provenance approaches developed with these systems are closed. These systems are in full control of the data they manage, and track their provenance within their own scope, but not beyond. However, there are few exceptions, discussed in [9] that begun to emerge with techniques to track provenance beyond their scope, but these tend to be ad-hoc solutions to specific problems.

Despite this body of research, capturing and presenting lineage information in script-based processing is still a challenging one. In databases the common assumption is the availability of manipulation operations that are limited to a well founded set, i.e., a collection of relational algebra operators or data replication primitives. Workflows can also be described as a detailed specification of a process, or as a set of interdependent data transformations. Therefore lineage can be captured by observing a workflow execution.

The style of code that scientists write differs greatly from code written by professional programmers. Scientists usually provide a long list of analysis scripts together with input data for processing. The black-box nature of these scripts limits the specificity of the lineage information that can be captured. This therefore makes it very hard to use existing lineage tracings mechanisms with such ad-hoc processing. To capture lineage for such processing, we designed a data model in Astro-WISE and then allow the users to extend the model and add any analysis routines. The model includes all APIs for connecting to the storage and processing nodes and also includes mechanisms of transparently recording and linking all dependency information. Although one of our goals is tracing lineage, our other goal is the development of a ‘machine learning’ kind of system which uses recorded and memorized history processing events for efficient processing of data sets.

The key contribution of this paper is to recognize that data lineage is also a *first-class data* and that data lineage can be used to simplify and partially automate the scientific data analysis cycle. The data analysis cycle is a process that requires expertise in techniques for scientific data analysis and the domain of the data being explored. This framework enables the effective reuse of data

lineage to aid both expert and non-expert users in performing scientific data analysis. The framework consists of two key components: we propose a new data lineage model that uses object oriented techniques and database support for persistent objects to uniformly capture detailed lineage data during the course of scientific data analysis cycle, then we also show that this detailed history information, when integrated into to a system simplifies the exploration process. It allows users to navigate through a large number of pipelines and data, giving them the ability to compare different pipelines and select an analysis pipelines, modify parameters and proceed with the scientific data analysis.

By means of full backward chaining we leverage on lineage data for processing, retrieving and archiving of data. We adopt a *pull-based* processing, where a user specifics 'a result' and the system automatically follows data lineage in the system to select and/or build the pipeline to process the result. The result will only be computed if it does not exist or only carry out incremental processing in case some of the input data already exists or needs to be recomputed. This framework also provides a scalable and easy-to-use interface where a user can modify the selected pipeline to customize or re-derive his own results following his insights.

The remainder of this paper is organized as follows. In Section 2, we provide a description of the design features of lineage tracing in Astro-WISE. We show in Section 3 the building-blocks of implementing the lineage features. In Section 4, we present the implementation of the lineage framework. We show how lineage is used for scientific processing in Section 5. Related work appears in Section 6 and we conclude in Section 7.

2 Data lineage in Astro-WISE

Astro-WISE[1] [3, 15] is a pipeline-centric data analysis system with the following features: Dependency tracking of both control-flow and data; command-line tools and an interface for exploring and sharing dependencies; use of dependency information to create analysis pipelines and a seamless interface with `Python`. Astro-WISE keeps track of all inputs, outputs and changes to a pipeline during a data-flow. This is includes all files used/created, objects, parameters, attributes, events and the run-time environment. In this section, we provide a description of the design features of lineage tracing in Astro-WISE.

2.1 Objects in Astro-WISE

Astro-WISE uses the advantages of Object-Oriented Programming (OOP) to process data in the simplest and most powerful ways. In essence, it turns

[1] www.astro-wise.org

data into OOP objects, called process targets, that are instances of classes with attributes and methods that can be inherited. The code for Astro-WISE is written in Python, a programming language highly suitable for OOP. Consequently, Python classes are associated with the various conventional calibration images, data images, and other derived data products. For example, in Astro-WISE, bias exposures become instances of the `RawBiasFrame` class, and twilight (sky) flats become instances of the `RawTwilightFlatFrame` class. These instances of classes are the objects of OOP.

Classes may have incorporated methods and attributes. Methods perform a task on the object they belong to, while attributes are properties such as constants, flags, or links to other objects that may be needed by methods. There may be different ways to create an instance of a class depending on which attributes are set to what values, and which methods are used. A `ColdPixelMap` object, for example, can be instantiated from the database (i.e. as the result of a query or search) or it can be created by using its make() method. The `make` process is synonymous to the unit/linux make metaphor. Each class defines a make method that specifies file targets and their dependencies, and commands transforming one to another for making a derived data product. For the remainder of this paper, the class names of objects, their properties, and methods (these latter are usually followed by () to identify them) will be in `typewriter text` font for more clear identification.

Every object is identified uniquely by an object identifier and a `version number`. During a dataflow, Astro-WISE keeps track of all objects used from a time a process is created to the point when execution ends. During processing, data files may be created and stored on a data-sever. Each data file in Astro-WISE is identified uniquely by a canonical name and this unique name is stored in the database as part of the metadata. In Astro-WISE the class which represents objects that have an associated data file is called a `DataObject`. Every instance of class `DataObject` or every class which is derived from class `DataObject` has an associated data file. The `FileObject` class is used to store the associated data files. The object identifier and the object type of a `DataObject` are used as reference to identify the relationship between the data file and the `DataObject`. Because of this relationship, Astro-WISE is able to track all files that have been used (or referenced) during a dataflow.

Astro-WISE does not allow objects to be modified. This is to enable accurate history tracking. Trying to store an object with the same object identifier will fail.

2.2 Dependencies in Astro-WISE

Astro-WISE makes use of the database for both input/output data. The database is part of Astro-WISE and as a result without access to the database, invoking a process on an object would fail. Astro-WISE tracks all parameters used (or created) runtime that induce any dependency relationship between objects, i.e., parameters that would affect the state of an object. These

relationships are the links that allow Astro-WISE to deduce the lineage of an object.

A dependency relationship is specified by *source object*(s), *modules* used during the processing, and *the sink object*. e.g., a function may depend on file inputs (the source object) and create file outputs (the sink objects), and use other functions. To eliminate false dependencies, Astro-WISE does not allow objects to be modified, neither does it allow an object to be deleted, if some objects have references to the object. However a dependency might be modified and stored as a new object. Even in such a case a reference to the old object is still maintained.

2.2.1 Control flow dependencies

These are dependencies for which a function (or module) directly affects the execution of another function or a function calls another function during a data-flow. Although using static-code analysis such information can be extracted, conditional statements, input data/parameters and the availability of intermediate may change the flow of processing. When intermediate data exists, the process using the intermediate data will inherit all its dependencies. Therefore accurate control flow dependencies can only be determined at runtime.

2.2.2 Data dependencies

The second category of events is those for which a process affects or is affected by data or attributes associated with an object. For example, processing parameters that could be changed during runtime or changing the input data to a process. Such dependencies allows Astro-WISE to store more fine grained data like processing parameters, object attributes and probably the run-time environment (which is often necessary especially for floating point computations).

2.3 Lineage graphs

By logging all dependency-causing events and their relationships during runtime, Astro-WISE uses the logged information to build a graph that depicts the dependency relationships between all objects that were involved in an execution. Such a graph is called a *lineage graph*. Using where-provenance described in [4] lineage-graphs can be visualized as bipartite graphs linking parts of the output with parts of the input.

The part of the Astro-WISE that builds such a lineage graph is called *ObjectViewer*. To construct the lineage (or dependency) graph, the *ObjectViewer* follows dependencies, starting from the object and builds a lineage graph recursively to the last dependency.

2.3.1 The data-lineage network

Knowing the lineage-network supports dependency exploration, pipeline building/extraction/sharing, and efficient processing through pipeline-reductions. Dependency exploration enables users to query the dependency network to understand relationships between objects or what events led to the creation of an object. Efficient processing through pipeline-reduction involves the use of a 'smart' updating engine that queries the dependency network to support incremental re-computation. With pipeline-reduction, a minimal set of control flow dependencies are executed especially if intermediate data exists. Before the processing of a target, the pipeline for execution is selected and dependencies analyzed by examining version of targets relative to their dependencies to determine those in need of updating (re-making).

3 Tracing lineage in Astro-WISE

Astro-WISE's processing is specified in terms of targets where each target knows how to process its self. This gives us the ability to trace the services that were used to generate these data products. In the case where services changes, a timestamps attribute is used for temporal ordering of the services in addition to causal ordering that may be implicit. During the making of a target, only newer dependencies are used. Newer dependencies are obtained by analyzing timestamps of targets/services together with version information if available.

The tracing and the use of lineage in this work are tightly integrated together. This integration enables Astro-WISE to trace accurate lineage and use the same traced data for processing. We ensure that all input/output data makes use of the database and therefore the lineage traced includes both the scientific processes and the database manipulations. To implement lineage tracing, we exploit four properties in a database and OOP environment.

- everything in Astro-WISE is an object. Classes in Astro-WISE are themselves objects;
- we apply the principle of inheritance, where all Astro-WISE objects inherit key properties for database access, such as persistence of attributes;
- the linking (associations or references) between instances of objects in the database is completely maintained;
- the continuous growth of the database through the addition of new information or improvements made to existing information.

3.1 Persistent objects

An object is said to be persistent if it is able to remember its state across program boundaries. This concept should not be confused with the concept of a program saving and restoring its data (or state). Rather, persistence, implies that object identity is meaningful across program boundaries, and can

be used to recover object state. Persistence is usually implemented by an explicit mapping from (user-defined) object identities to object states and by then saving and restoring this mapping. However, this implementation assumes that the object identity of the object a user is interested in can be independently and easily obtained. For many applications this is not the case. On the contrary, one usually has a (partial) specification of the state, and are interested in the corresponding objects that satisfy this specification. That is, many interesting applications depend on a mapping of a partially specified object state to object identity (and then to object). This is the domain of the relational database.

A relational database management system (RDBMS) stores, updates and retrieves data, and manages the relation between different data. A RDBMS has no concept of objects, inheritance and polymorphism, and it is therefore not a-priory obvious that one would like to use such a database to implement object persistence. However, using the following mapping

$$\begin{aligned} \text{type} &\longleftrightarrow \text{table} \\ \text{identity} &\longleftrightarrow \text{row index} \\ \text{state} &\longleftrightarrow \text{row value} \end{aligned}$$

we can implement object persistence using a relational database. That is, given a type and object identity, one can store and retrieve state from the specified row in the corresponding table. Relational databases provide a powerful tool to view and represent their content using structured queries. We leverage this power to efficiently search for object whose state matches certain criteria. Special consideration has to be given to inheritance in this case. Assume, for example, that we define a persistent type `DomeFlatImage`, derived from a more general type `FlatfieldImage`. A query for all R-band flatfield images, should result in a set including all R-band domeflat images. This behavior of queries is what inheritance means in a relational database context. Hence, a query for objects of a certain type maps to queries (returning row indices/object identities) on the tables corresponding to that type, and all of its subtypes. The results of these queries are then combined in to a single set of all objects, of that type or one of its sub types, that satisfy the selection.

3.2 The persistent object hierarchy and object linking

Figure 1 shows the various components of the Astro-WISE pipeline. There are three layers.

- The Library layer provides a number of low level interfaces for data processing, a persistence mechanism with database interface(s), and a set of auxiliary libraries, many of which are part of the standard Python library.
- The API (Application Programmers Interface) layer consists of two components. The Persistent Object Hierarchy, processing steps are performed by calling methods on these objects. Since these objects are persistent, all operations and values assigned to attributes of these objects are

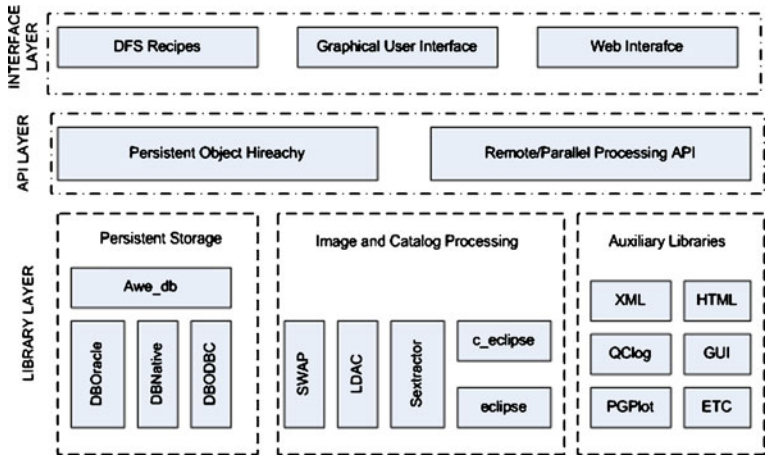

Fig. 1 Components of Astro-WISE's pipeline

automatically saved into a database. The API layer also provides an interface for parallel/remote processing.
– Finally, the Interface layer, provides user interfaces, or applications, to invoke and manipulate the pipeline processing and an interface to view and query for lineage data. A user interacts with the library through the persistent object layer.

For each persistent class G, a mapping table is defined, whose columns represent the persistent attributes of G and whose rows represent all objects of type G. A persistent property is defined by using the following expression in the class definition.

```
prop_name = persistent(prop_docs, prop_type, prop_default)
```

where, `prop_name` is the name of the persistent property, and persistent is constructed using three arguments: the property documentation, the type of the property, and the default value for the property respectively. We distinguish between 5 different types of persistent properties, based on the signature of the arguments to persistent.

– **descriptors**: If the type of the persistent property is a basic (built-in) type, then we call the persistent property a descriptor. Valid types are: integers (int), floating point numbers (float), date-time objects (datetime), and strings (str).
– **descriptor lists**: Persistent properties can also be homogeneous variable length arrays of basic built in types, called descriptor lists. Valid types are the same as those for descriptors. Descriptor lists are distinguished from descriptors by the property default. If the default is a Python list, the the property is descriptor list, else it is a simple descriptor.

- **links**: Persistent objects can refer to other persistent objects. The corresponding properties are called links.
- **link lists**: Persistent properties can also refer to arrays of persistent objects, in which case they are called link lists. Link lists are distinguished from links by the property default. If the default is a Python list, the the property is link list.
- **self-links**: A special case of links are links to other objects of the same type. These are called self-links. if no type and default are specified for the call to persistent, then the property is a self-link.

3.3 Database architecture

Access to the database is provided through `Python`. The database interface maps class definitions that are marked as persistent in `Python` to the corresponding `SQL` data definition statements. The database stores all persistent objects, attributes and raw data either as fully integrated objects or as descriptors. Only pixel values are stored outside the database in image and other data files. These data files are registered in the Astro-WISE metadata database with the unique filename which can be used to retrieve the data item from the system. Along with the name of the file, the data model for the item is stored in the metadata database. This includes all persistent attributes specified by the model.

Data can therefore only be manipulated through interaction with the database. A query to the database will provide all information related to the processing history and to the locations of all stored associated files, attributes and objects.

3.4 Extendable schema

The database provides an extendable schema that allows extending and modifying object attributes and method definitions through inheritance and polymorphism. This then allows end-users to define new persistent data products. Because schema modification complicates history tracking, we support schema evolution through inheritance and polymorphism to introduce new classes (or processing routines) to the pipeline. That way, any new processing routines can be defined and added. Users can modify functionality in modules, insert them into the system, or add a module on top of what currently exists, as long as these modules obey the standard data model. However allowing the schema to be modified may create object inconsistencies (mismatches). A mismatch is when the retrieving system tries to retrieve a particular object whose own generating class was different in the storing system, and whose structure is inconsistent with what the retrieving system is expecting.

Although research prototypes propose a strict conversion [11] of data to a new format, history tracking is lost when old object ceases to exist. Astro-WISE on the other hand is designed to compare versions of data and classes. If there is a newer version of an object or a class, all dependent objects

become outdated. A request for an outdated object will cause the object to be computed on-the-fly.

4 Implementation

To implement lineage tracing, we implement the base class that gives derived classes the property of being persistent. We also implement the database interfaces that support schema evolution and querying. We define a persistent class hierarchy derived from the persistent base class, that supports the Astro-WISE's data model. This includes an implementation for all data processing routines as methods on persistent objects.

In an object-oriented programming language, you can define classes, whose purpose is to bundle together related data and behaviors. These classes can inherit some or all of their qualities from their parents, but they can also define attributes (data) or methods (behaviors) of their own. Classes generally act as templates for the creation of instances. Different instances of the same class will typically have different data, but it will come in the same shape.

In `Python` classes are themselves objects that can be passed around and introspected. Since objects, as stated, are produced using classes as templates, a metaclass acts as a template for producing classes. If we subclass the original metaclass 'type' we can dynamically generate and control behavior of all classes. There are two general categories of programming tasks where we think metaclasses are genuinely valuable.

The first, and probably more common category is where you do not know at design time exactly what a class needs to do. Obviously, you will have some idea about it, but some particular detail might depend on information that is not available until later. "Later" itself can be of two sorts: (a) When a library module is used by an application; (b) At runtime when some situation exists. Secondly, with metaclasses we can add, delete, rename, or substitute methods for those defined in the produced class.

We used the *metaclass* (i.e., `DBObjectMeta`) to define properties that are common to all classes. All classes derived from `DBObjectMeta` will inherits a set of features and these will be available to all classes. One of these feature is the persistent property. So all classes are *persistent* classes, implying the class properties and instantiations are automatically made persistent in the database as conceptually shown in Fig. 2. The complete framework is implemented by the following major classes;

- `DBObject` is the root class of the hierarchy of the persistent classes. This class defines the primary key `object_id` of all objects.
- `DBObjectMeta` is the metaclass of `DBObject`. This is the class that is responsible for class creation and object instantiation.
- `DBProperty` is the module that defines all persistent attribute types which are defined by `persistent`.

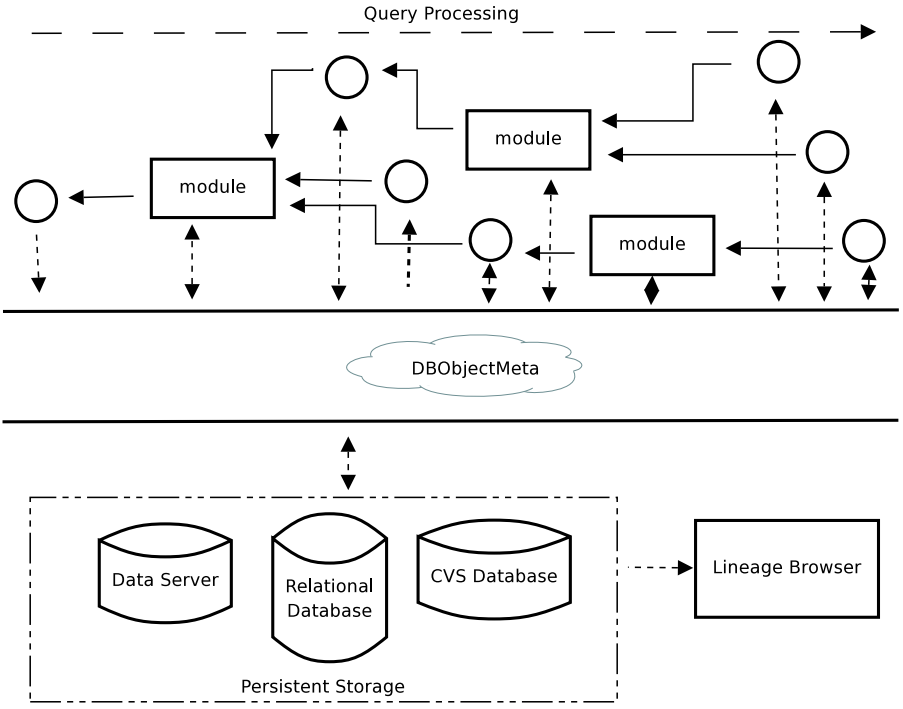

Fig. 2 The figure shows the interaction between the *Target processor* and *DBObjectMeta* class which controls persistent class creation and delegates persistent object instantiation to the database. The *solid lines* represent the flow of processing, the *dashed arrows* shows data movement to and from persistent storage, the *circles* represent input/output data to a service and the *boxes* represent the different services

- `Deselect` implements a query language that is a natural extension to `Python` and that incorporates data lineage in the query syntax.
- `DBProxy` is an abstract interface to database vendor specific operations.

`DBObjectMeta` defines two special methods `__new__()` and `__init__()`. The `__init__()` method lets you configure the created object; the `__new__()` method lets you customize its allocation. Specifically to Astro-WISE, the `__new__()` method creates new instances of `DBObjectMeta`, which are new class objects (i.e., persistent classes). This method is called at class definition time. This method analyses the attributes and instantiates specific persistent property objects from `DBProperty` for all attributes that are of type persistent.

4.1 Lineage capture and storage

Data lineage is captured as dependencies between persistent objects that refer to another persistent object. Using the implementation above and the

definition/description of persistent properties in Section 3.2, we provide some examples;

Example 1

```
class ClassA(Object):
        a = persistent('', int, 0)
        b = persistent('', float, [])
        c = persistent('', str, ['A', 'B', 'C'])
```

ClassA in Example 1 defines three persistent attributes:

- 'a' is an int,
- 'b' is an array of floats, with an empty default,
- 'c' is an array of strings with a three element default.

Example 2

```
    class ClassB(Object):
        e = persistent('', ClassA, (ClassA, (), {})
        f = persistent('', ClassA, [])
        g = persistent('')
```

ClassB in Example 2 defines three links:

- 'e' is a link to an instance of ClassA,
- 'f' is a array of links to instances of A (default empty), and
- 'g' is a link to another instance of ClassB

Note that persistent properties are inherited. So ClassC in Example 3 defines a new persistent object, with four persistent attributes ('a', 'b', 'c', 'h').

Example 3

```
    class ClassC(ClassA):
        h = persistent('', ClassB, None)
```

To support the storage of files in a similar way, the `DataObject` class is used as shown in Example 4

Example 4

```
class DataObject(DBObject):
  filename = persistent('File part', str, '')
```

The `filename` attribute is used to implement `store()` and `retrieve()` methods to transfer files to and from data-servers. Because the filename is

kept in the database the storage and retrieval of files is transparent to the application. In the Example 5 the file `myfile.fits` is made persistent by storing the file and committing the object of which it is part. Then we can search for all the files with a name that starts with `myfile.fits` and fetch the first one (query[0])found from the data-servers.

Example 5 Storing a persistent file

```
myobject = DataObject(pathname='myfile.fits')
myobject.store()
myobject.commit()

query = DataObject.filename.like('myfile*')
query[0].retrieve()
```

The definition of links above makes all objects persistent in the database forming a dynamic archive of all targets. Since persistent objects references to (instances of) other persistent objects. We have ensured that instantiation of a persistent object does not recursively instantiates all objects it refers to. Only when the attribute corresponding to the reference is accessed then the corresponding object is instantiated. e.g., consider `object = ClassB()` (from Example 2). `print object.e` will only result in instantiation of a new `ClassA()` when attribute 'e' is accessed.

4.2 Processing parameters

Many processes require parameters. Default values for these parameters are set in the code. Parameters are themselves persistent objects, so that we can always retrieve from the database the parameters that were used in a particular process. Parameters can be changed by:

1. Making new instances of parameter objects (with new values) in user-defined scripts
2. Providing new parameter values through a user Interface

Note that the user does not change parameters by changing the default value that is supplied in the source code, but by providing alternative values in their own scripts.

All derived objects, e.g., conventional astronomical calibration images/products, are collectively referred to as "process targets" and inherit from the `Process` class. Each `Process` class has an associated processing parameters object, an instance of a class named after the respective process target class which stores configurable parameters that guide the processing or (re)processing of that target. Those processes that use external programs in their derivation may have additional objects associated with them which

contain the configuration of the external program that was used. Below is an example,

```
DarkCurrent
 |
 +--cosmicconf
 |   |
 |   +--ANALYSIS_THRESH: 100.0
 |   +--BACKPHOTO_THICK: 24
 |   +--BACKPHOTO_TYPE: GLOBAL
 |   +--BACK_FILTERSIZE: 3
...
 +--process_params
 |   |
 |   +--DETECTION_THRESHOLD: 6.0
 |   +--MAXIMUM_DARK_CURRENT: 5.0
 |   +--MAXIMUM_DARK_CURRENT_DIFFERENCE: 0.2
 |   +--MAXIMUM_ITERATIONS: 5
 |   +--OVERSCAN_CORRECTION: 6
 |   +--REJECTION_THRESHOLD: 5.0
```

Where the `cosmicconf` tree refers to the configuration of the external program and the `process_params` tree refers to the process parameters of the `darkcurrent` object.

4.3 Control-flow dependencies

Using the metaclass (see Section 4), we can add, delete, rename, or substitute methods for those defined in the produced classes. In the same way, we decorate all make methods of all Astro-WISE classes, this enables us to trace and log all method calls. We keep track of the call-hierarchy graph and for each method executed at the top of the `Python` stack, the method called, method arguments and return values are logged to the provenance store. From this information, we have a trace of all method calls specifying how they were invoked, as well as what arguments were used for each method. An example of such a trace is shown in Log 1. The log shows the methods (`__init__`, `set_persistent_parameters`, `get_kw_list`) that were called for the `AssociateConfig` class, how they were called and the input parameters and return values for each method.

Log 1 A control-flow dependency

```
AssociateConfig.__init__(filename=t-g8VR.associate.conf)
   AssociateConfig.set_persistent_parameters()
   : None
   AssociateConfig.get_kw_list()
   : [('INTER_COLOR_TOL', 2.5), ('ISO_COLOR_TOL', 1.5),
   ('MASK', 216)]
: None
```

4.4 Querying lineage

We have defined a notation that is based on the idea that a class is in some sense equivalent to the set of all its instances. To illustrate the concept, let us give a few examples. Given a persistent class `ReducedScienceFrame` with persistent property `Raw`, then the expression `ReducedScienceFrame.Raw` = *value* represents the set of all instances `reduced` of `ReducedScienceFrame`, or subclasses of `ReducedScienceFrame`, for which `reduced.raw` = *value* is true. To obtain these objects the expression needs to be evaluated, which can be done by passing it to a select function, which returns a list of objects satisfying the selection. More generally, given a class X with a descriptor `desc`, a descriptor list `dsc_lst`, and a link `lnk` then,

```
select(X.desc > 2.0 && X.dsc_lst[2]= abc and X.lnk.attr = 5)
```

will return a list of instances x of X, or subclasses of X, for which `x.desc > 2.0` and `x.dsc_lst[2] = 'abc'` and `x.lnk.attr = 5` is true.

5 Results

When an astronomer identifies a certain object (in a source list or in an image), the astronomer should be able to trace any bit of information that was involved in the derivation of that source and the astronomer should be directed to all available information within the database relevant for the same sky position. Sometimes it is necessary to compare results of images in several filters or epochs with respect to that sky positions to have a quick look at the properties of any object. This functionality will form the core of tracing history and relations to other data items, which will also be used for inspecting and triggering target processing (or on the-fly-reprocessing). We have developed tools to query and retrieve all lineage/provenance data from the Astro-WISE database through a common interface. These tools display everything related to a certain object. Everything means all data (or pointers to that data) and software that was used to create the object, and that which might be used to recreate the object with a different setting.

5.1 Using data lineage

We demonstrate the use of lineage in Astro-WISE in two ways; Firstly, we integrated lineage into the processing machinery of Astro-WISE to simply processing of data through target processing (detailed in Section 5.1.1). This integration simplified and automated data analysis in Astro-WISE such that both expert and non-expert users can perform data analysis effectively. Secondly in Section 5.1.2, we show how lineage data can be used to explain differences between two objects by comparing derivation lineage graphs of two

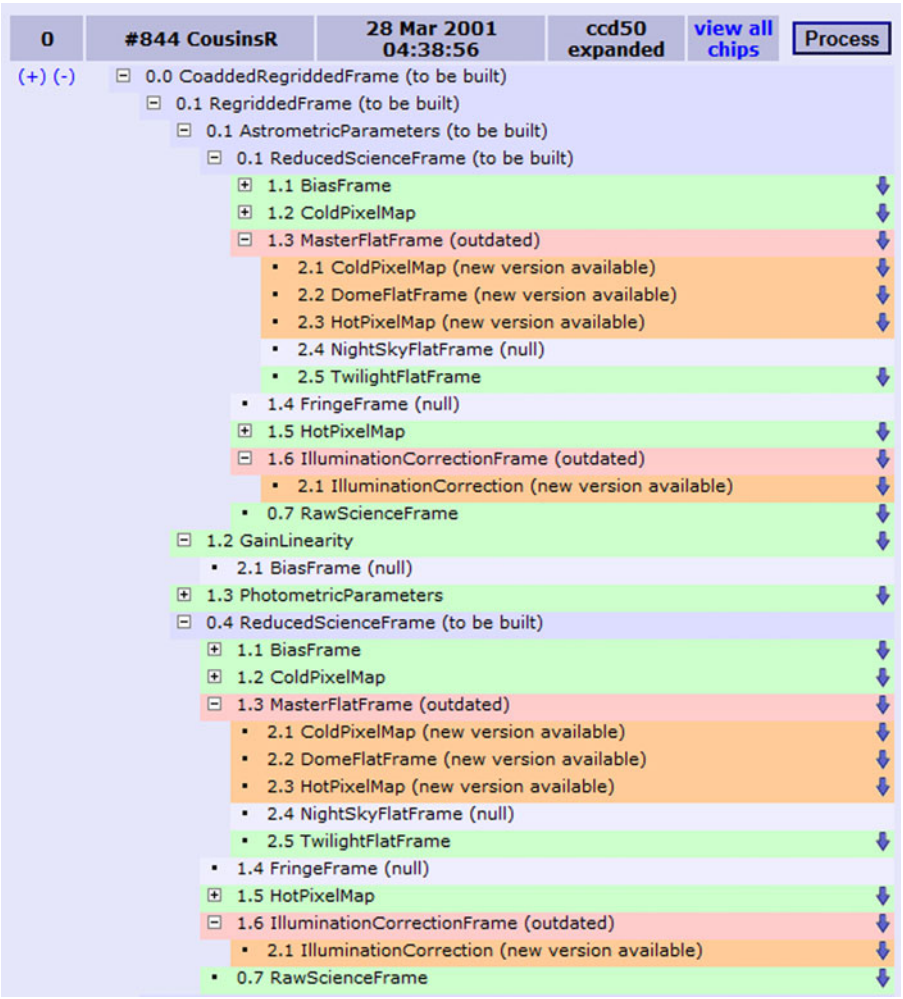

Fig. 3 Target processing: a tree view is given of the target(s). This tree view gives an overview of the target dependencies. *Green* dependencies are up-to-date, *red* dependencies are out-of-date and for *orange* dependencies indicate a newer version exists

objects, this can be automated by providing algorithms that can difference two derivation trees.

5.1.1 Target processing

Astro-WISE follows a '*pull-based*' i.e., backward-chaining, approach while processing data. This allows the end user to trace the data product, following all its dependencies up to the raw observational data and if necessary, to re-derive the result with better data, and/or improved methods. This paradigm is characterized by a fixed set of homogeneous, well-documented data products.

This is unlike *push-based*' i.e., forward-chaining systems, where the end user has little or no influence on what happens upstream, operators have a task to push the input data through the stream, often by means of a pipeline, irrespective of whether the derived data items are actually used by the end users.

A target is a database object representing a file or metadata that is passed as input to and generated as output from a process. Specifically in astronomy, a target could be any calibrated image, or the results for a set of calibration parameters, or a list of parameter values describing an astronomical object. For example, one of the dependencies in processing science data is the `flatfield`. A `flatfield` can be selected from the database based on the date of observation of the science data and the validity time-stamp of flatfields in the database or can be reconstructed on the fly if there exists significant improvements in the code.

The target processor is a web based target processing engine for Astro-WISE. A pipeline for processing a target is specified in terms of targets and dependencies. To construct the necessary pipeline, the target processor employs a dependency logic derived from data lineage. The basic idea behind the target processing is to construct a graph representation of all targets required to make a target. Where each node in a graph is its self a target and the edges represent dependency relationships. Each target (object) knows how to process its self. Each target has a `make`[2] method which explicitly specifies file targets, their dependencies, and modules/methods required to process the target.

Targets have dependencies that may themselves be targets. Targets are built, in a recursive cascade, only when their dependencies have changed. By maintaining dependency links between modules (source code) and data generated by each module, each module is aware of the targets associated with the preceding invocations of the module or any previous runs of the module.

These features allow the target processor to use existing dependency information (results) instead of recreating them. The existing data is analyzed for changes in dependencies or for newer versions. Based on the results of the analysis, the target processor will reduce the processing tasks to only those required to create the target (with newer dependencies). Figure 3 shows the final graph representation required to make a target. Each node (target) in the graph has been analyzed and marked with a colour code. The colour codes indicate the targets that will be processed and those that will be used as-is. In some extreme cases the target processor may return the final data product if it already exists (i.e. processed before) and is up-to-date (i.e., none of the dependencies has changed) without any processing.

5.1.2 Comparing `DataObjects`

During a scientific study, users often have to integrate new features into existing pipelines. For example, a user may wish to improve a given result by

[2]The `make()` method follows the `linux` make metaphor.

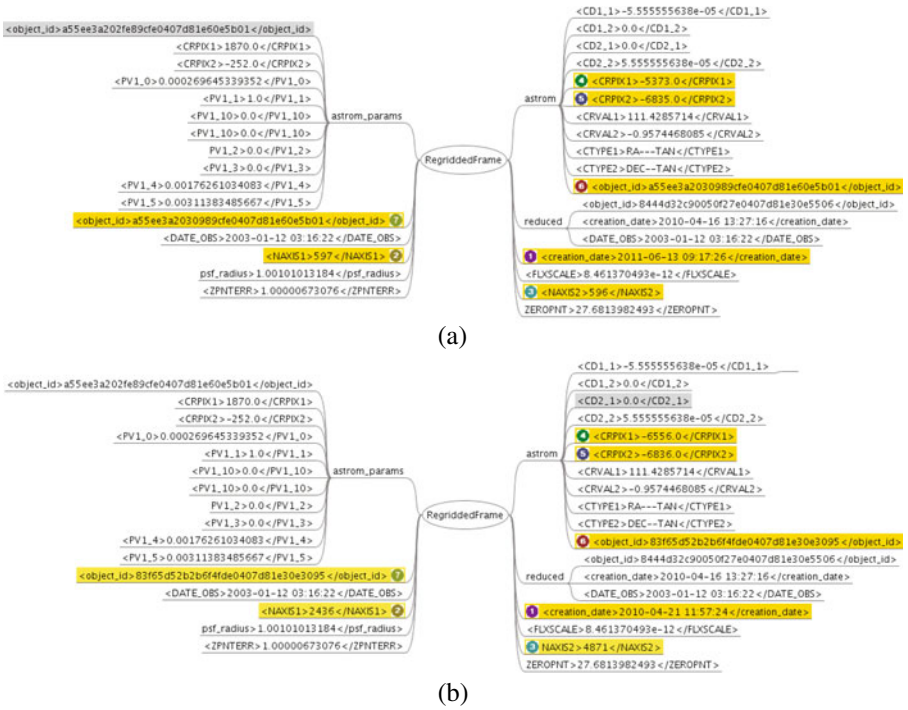

Fig. 4 Lineage graph of two `RegriddedFrames`. The *nodes highlighted* are the differences between both objects

adjusting parameters, attributes or sometimes switch to a different algorithm. All this information is recorded during the study and is linked to the object created. By comparing this information we can explain differences between two objects (or two runs). In Fig. 4 we compared two `RegriddedFrames`, that were created from the same `ReducedScienceFrame`. During the process of creating the `RegriddedFrames` some parameters were changed. These changes are marked and highlighted in both figures. Each `RegriddedFrame` is identified with a unique object id. From the figure, both frames used the same `astrom_params` and `reduced` objects. The complete lineage for both objects could not be displayed due to space limitations. Such a feature could help a user analyze, compare and understand the differences between objects.

5.2 Data usage and rates

For everything processed in Astro-WISE, we store all images generated during the data flow and all processing data to centralized database. Using this information we can quickly compare how many times an object has been reprocessed or how many objects refer to it. For example, Fig. 5 shows the number of versions of BiasFrames that were derived from a fixed set of RawBiasFrames as a function of time. The example output shows that different

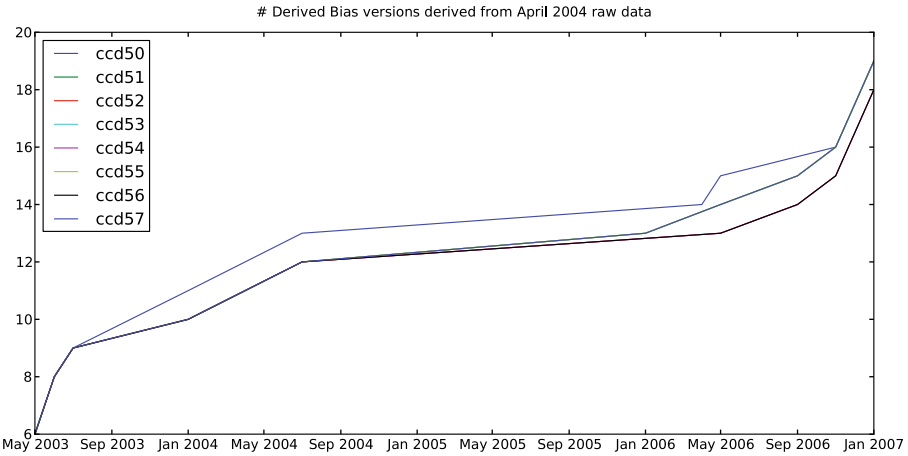

Fig. 5 Graph showing the data growth rate, from the point an image is ingested into the system

CCD's (ccd50–ccd57) have been reprocessed different number of times, even though they belong to the same image. The y-offset of 6 at the beginning means that six raw images are considered in this sample.

Input datasets to pipeline in eScience is usually very large in size. These datasets are too large to be transferred efficiently via the Internet. Owing to bandwidth limitations of the Internet. However the usage analysis could reveal data usage patterns and this can be useful to determine a cost-effective storage strategy, which could significantly reduce the total processing cost on all data products that depend on such data objects.

5.3 Viewing lineage data

Astro-WISE's data lineage service provides several interfaces of querying and viewing lineage data. Three specific interfaces are presented in this paper:

1. the web interface object viewer tool;
2. the dependency cutout web service;
3. using `Python` scripts which are translated into SQL statements automatically;

To reduced on data overload, an additional parameter is specified on how deep the query trace should recurse in the case where dependencies contain nested dependencies.

5.3.1 The object viewer

The object viewer queries and displays all history processing information for an object from the Astro-WISE database. The algorithm for constructing the lineage graph is similar to an algorithm used to construct a graph given

the set of node-edge pairs. Beginning from the object, whose lineage is to be displayed, the object type specifies the the entity table to be looked up. Using the object ID attribute all objects attributes in the table are identified. These attributes lead to the data produced and used (e.g., image data or processing parameters) by that object. This available through links referring to other attributes or database objects. This is followed recursively till the last dependency.

The object viewer will show everything related to a derivation of a an object. A lineage tree-like view of a *SourceList* with ID 10011 is shown in Fig. 6. In the tree, leaves correspond to attributes that have an atomic type and branches correspond to links/references to objects. By expanding a branch one can have a more detailed look at the referenced object.

5.3.2 Dependency cut-out service

This service provides methods to create cutouts of a requested size from the all dependant images (i.e. images generated or used in the processing of an object). The service follows lineage in the system to find all dependant images

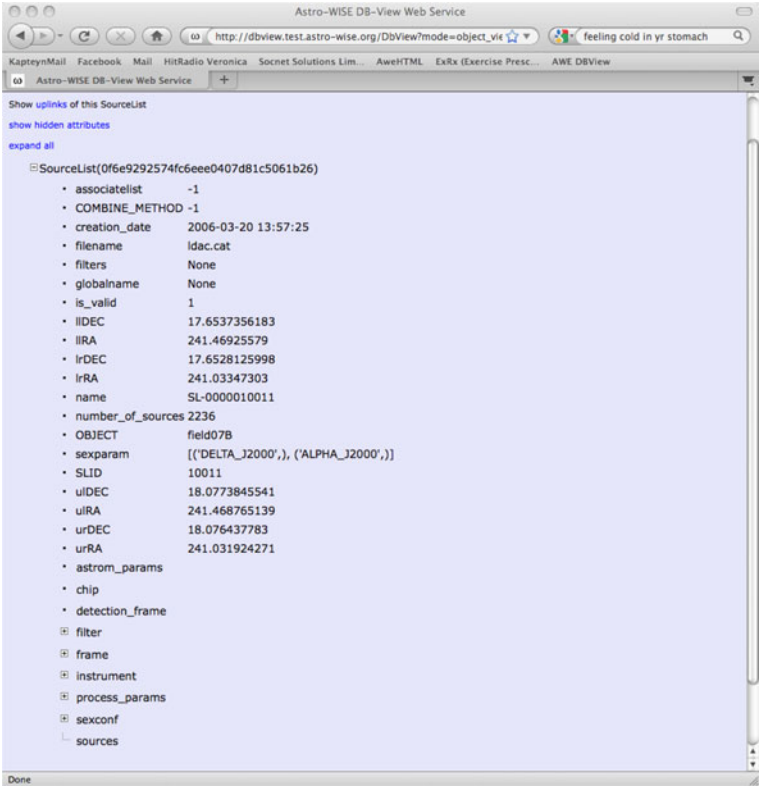

Fig. 6 Web interface showing all dependencies of an object

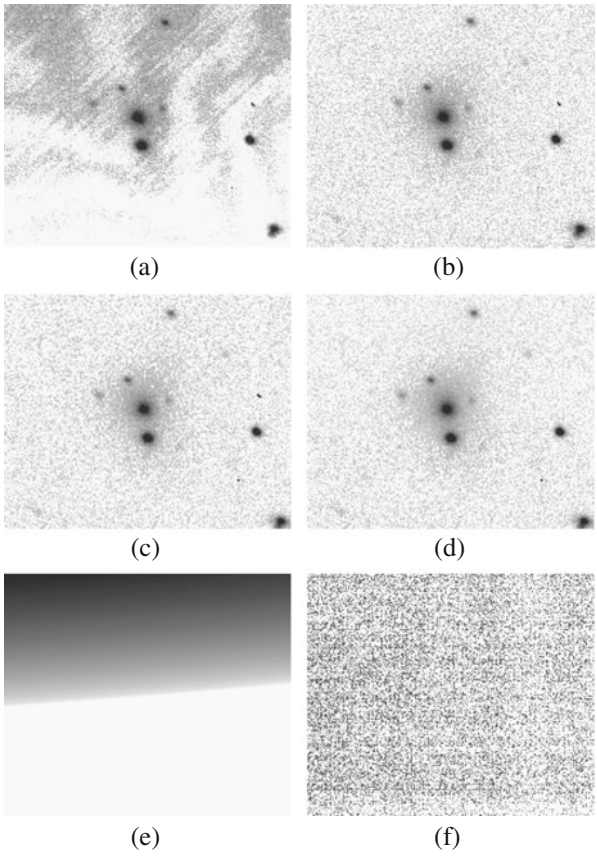

Fig. 7 The images above show all image cutouts of one of the sources. The **a** RawFrame, **b** ReducedFrame, **c** RegriddedFrame **d** CoaddedRegriddedFrame were used in the Image Pipeline and **e** Illumination Correction and **f** MasterBias were used in the calibration pipelines during the processing of SourceList

and the actual location of the object on the dependant images. Figure 7 shows an example of image cutouts of some of the images that were created or used during the processing of one of the sources.

5.3.3 Scripts for lineage retrieval

The provenance graph is a relational structure, and as such there are a wide variety of languages available for querying the graph, ranging from simple path or reachability queries, to SQL-like relational queries, to more expressive languages supporting recursive queries. To this end, we implemented an extended query language which is a natural extension to `Python`. The structure of dependency queries is of the form.

```
ClassA.link_propertyB.link_propertyC.filename.
```

This is translated into an SQL query and sent to the database for execution. Take an example a

```
query = RegriddedFrame.swarpconf.RESAMPLING_TYPE == 'LANCZOS3'
```

is translated into

```
WITH SQ20 AS (
 SELECT "object_id" AS "swarpconf.object_id"
    FROM AWOPER."SwarpConfig+"
 WHERE "RESAMPLING_TYPE" = 'LANCZOS3'),
 SQ21 AS (
 SELECT "object_id" AS "chip.object_id","name"
                    AS "chip.name" FROM AWOPER."Chip+"),
 SQ22 AS (
 SELECT "object_id" AS "filter.object_id","name"
                    AS "filter.name" FROM AWOPER."Filter+"),
 SQ23 AS (
 SELECT "object_id" AS "instrument.object_id","name"
                    AS "instrument.name"
                    FROM AWOPER."Instrument+"),
 SQ10 AS (
 SELECT "+PROJECT" AS "project_id",'quality view'
    "quality view", "+PRIVILEGES","object_id",'re-process'
    "re-process",SQ23."instrument.name",
    SQ22."filter.name",SQ21."chip.name","creation_date",
    "DATE_OBS",'preview'"preview","filename","FLXSCALE",
    "globalname","is_valid","NAXIS1","NAXIS2","OBJECT#",
    "process_status","psf_radius","quality_flags","ZEROPNT",
    "ZPNTERR","OBJECT_ID$",SQ20.
    "swarpconf.object_id" FROM AWOPER."RegriddedFrame+"
    JOIN SQ20 ON "swarpconf@" = SQ20."swarpconf.object_id"
    LEFT JOIN SQ21 ON "chip@" = SQ21."chip.object_id"
    LEFT JOIN SQ22 ON "filter@" = SQ22."filter.object_id"
    LEFT JOIN SQ23 ON "instrument@"
      = SQ23."instrument.object_id"
    ORDER BY "creation_date" DESC)
 SELECT * FROM SQ10
    WHERE ROWNUM <= 100 AND SQ10."project_id"
                  = SYS_CONTEXT('AWCONTEXT','PROJECTID')
```

We also identified basic operations that are useful for common querying tasks over provenance store that simplify the query syntax. Some examples are listed below;

- `get_inverse_properties(obj)` returns all objects that used *obj*
- `get_dependencies(obj)` returns all attributes of *obj*
- `get_onthefly_dependencies(obj)` returns all processable dependencies which can be used to recreate an object on the fly.
- `info(obj)` displays a lineage tree for *obj*

– `get_persistent_properties()` displays all persistent properties of an *obj*

6 Related work

Several provenance models do exist in literature today. Systems such as StarFlow [7], Chimera [8], Kepler [1], Karma [13], and ZOOM [5] have developed mechanisms for provenance recording and retrieval. Common among these systems is, that provenance recorded represents a complete workflow execution during a data-flow. However all these systems are seldom accompanied by formal specifications of their intended semantics. As a result it is very hard to understand provenance information produced by a workflow system or use the same provenance model in another system. However, recently the Open Provenance Model [10] has been developed as a consensus exchange format for representing provenance graphs. The scarcity of clear specifications of the semantics and provenance behavior of provenance aware systems makes it difficult to use existing models and also to compare or evaluate provenance models. It would be interesting to compare and unify these different techniques so that a common provenance model is defined. We have used object oriented design and database support for persistent objects to trace lineage (provide provenance) of data and support e-science research. Although our approach differs from existing models, it does not necessarily replace existing methods, especially for those readers who are interested in only the derivation history of data. We introduce another concept in lineage tracing and use of lineage in e-science that may incite further research in this area.

The use of data lineage has started to attract the attention of scientists. We used data lineage to simplify the creation of pipelines and to avoid the re-processing of already derived data. Kepler [14] too, has the Smart Rerun Manager (SRM) to handle all the tasks associated with the efficient rerun of a workflow which was developed from the VisTrails cache management algorithm [2]. The Vistrails algorithm is used to search for a graph representation of the dataflow for sub-graphs that have been successfully computed before and the sub-graphs that must be rerun. The intermediate data products for the sub-graph are retrieved from the provenance store and are used to replace this sub-graph.

The precondition for selection of intermediate data for both systems is that intermediate data was created with the current parameters and input data. Astro-WISE goes beyond checking for only parameters, but also analyzes all dependencies of each intermediate data product. If a newer version of a dependency of an intermediate data product exists, e.g., dependencies that could have been made with new versions of code, the intermediate data product will be re-processed. Therefore, during the creation of our pipeline, we use both saved provenance data and also up-to-date results. Whereas in Kepler, the user has to choose between doing a smart rerun of the workflow

with saved provenance data to exactly recreate a past run or to rerun the workflow to get the most up-to-date results.

7 Conclusion

In this paper, we have described the Astro-WISE data lineage Framework. We have shown how Astro-WISE traces lineage of data, how it uses the same data for retrieving, archiving and processing of scientific data which has relevance to e-science data processing needs. Using object persistence, inheritance and polymorphism, users can extend functionality of the system, create, store and distribute links. The target processor, the main processing engine, follows the dependency logic and lineage of data while processing targets. This is facilitated by the full end-to-end linking of all dependent data items. This abstraction enables Astro-WISE to guide the user to that intrinsic information by forcing full backward and forward chaining in the data modeling. Experimentation has been done using real astronomical examples. These examples demonstrate that lineage information is highly effective in scientific processing and greatly benefits scientific users of the system.

Lineage traced in this work is coarse-grained. The model allows us to trace all input/output data (images) and all parameters that were used during a processing. However we have no knowledge about which pixels contributed to a final result. Further extensions to Astro-WISE is to consider the problem of lineage tracing at pixel level.

References

1. Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., Mock, S.: Kepler: an extensible system for design and execution of scientific workflows. ssdbm **00**, 423 (2004)
2. Bavoil, L., Callahan, S.P., Crossno, P.J., Freire, J., Vo, H.T.: Vistrails: enabling interactive multiple-view visualizations. In: IEEE Visualization 2005, pp. 135–142 (2005)
3. Begeman, K.G., Belikov, A.N., Boxhoorn, D.R., Dijkstra, F., Valentijn, E.A., Vriend, W.J., Zhao, Z.: Merging grid technologies. Journal of Grid Computing **8**, 199– 221 (2010)
4. Buneman, P., Khanna, S., Tan, W.C.: Why and Where: a Characterization of Data Provenance, vol. 1973, pp. 316–330. Springer (2001)
5. Cohen-Boulakia, S., Biton, O., Cohen, S., Davidson, S.: Addressing the provenance challenge using zoom. Concurr. Comput. Pract. Exper. **20**(5), 497–506 (2008)
6. Cui, Y., Widom, J.: Lineage tracing for general data warehouse transformations. In: VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases, pp. 471–480. Morgan Kaufmann, San Francisco, CA, USA (2001)
7. Elaine Angelino, D.Y., Seltzer, M.: Starflow: a script-centric data analysis environment. In: McGuinness, D.L., Michaelis, J.R., Moreau, L. (eds.) Provenance and Annotation of Data and Processes, Third International Provenance and Annotation Workshop, IPAW 2010, Troy, NY, USA, 15–16 June 2010. Proceedings. Lecture Notes in Computer Science, vol. 6378. Springer (2010). doi:10.1007/978-3-642-17819-1

8. Foster, I., Vockler, J., Wilde, M., Zhao, Y.: Chimera: avirtual data system for representing, querying, and automating data derivation. ssdbm **00**, 37 (2002)
9. Moreau, L.: The foundations for provenance on the web. Found. Trends Web Sci. **2**, 99–241 (2010). doi:10.1561/1800000010
10. Moreau, L., Freire, J., Futrelle, J., Mcgrath, R.E., Myers, J., Paulson, P.: Provenance and annotation of data and processes. In: The Open Provenance Model: An Overview, pp. 323–326. Springer, Berlin, Heidelberg (2008)
11. Penney, D.J., Stein, J.: Class modification in the gemstone object-oriented dbms. In: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, OOPSLA '87, pp. 111–117. ACM, New York, NY, USA (1987)
12. Reilly, C.F., Naughton, J.F.: Transparently gathering provenance with provenance aware condor. In: First Workshop on Theory and Practice of Provenance, pp.13:1–13:10. USENIX Association, Berkeley, CA, USA (2009). http://dl.acm.org/citation.cfm?id=1525932.1525945
13. Simmhan, Y., Plale, B., Gannon, D.: Karma2: provenance management for data-driven workflows. Int. J. Web Service Res. **5**(2), 1–22 (2008)
14. System, W., Altintas, I., Barney, O., Jaeger-frank, E.: Provenance collection support in the kepler scientific workflow system. In: In Proceedings of the International Provenance and Annotation Workshop (IPAW), pp. 118–132. Springer (2006)
15. Valentijn, E.A., McFarland, J., Snigula, J., Begeman, K., Boxhoorn, D., Renegelink, R., Helmich, E., Heraudeau, P., Kleijn, G.V., Vermeij, R., Vriend, W.J., Tempelaar, M.J.: Astrowise: chaining to the universe. In: Astronomical Data Analysis Software and Systems XVI, ASP Conference Series, vol. 376 (2007)